Problem 5 Editorial: Let's Get this Bread
4+2+2+1=9 Points

Problem ID: `bread`
Rank: `2+2+3+3`

Overview

Problem statement: https://calico.berkeley.edu/files/calico-fa22/contest/bread/bread.pdf

This was our first contest problem with non-cumulative bonus test sets! The main test allowed for direct brute-force approaches, while bonus test set A emphasized solution efficiency and bonus test set B emphasized solution flexibility. Solutions that solved bonus test set A could have made use of some sliding window or two-pointer techniques to speed up calculations; solutions able to solve bonus test set B commonly used recursion (with memoization) or dynamic programming to account for the use of multiple meal cards. Bonus test set C brings both these ideas together; it's possible to piece together solutions from the two other bonus test sets (with minimal additional work!) to form a solution that has the best of both worlds!

Note that throughout this editorial, days in the semester are one-indexed. Namely, "day 1" refers to the day the problem begins with, and "day $\mathbf{N}$" refers to the last day in the semester.

Main Test Set

With only 1 meal card and up to 50 days left in the semester, this test set is relatively generous in terms of the efficiency it requires!

ALL the Intervals *Solution available in Python*

With just one meal card to use during the semester, we can simulate how much bread we can eat if we swiped on each possible day—this is called a complete search! For each day of the semester, simulate the result of using a swipe on our current day—for day i, this involves us summing the amount of bread available at the cafeteria $\mathbf{B}_i + \mathbf{B}_{i+1} + \ldots$, stopping after $\mathbf{D}$ days have passed (in other words, we've added $\mathbf{B}_{i+\mathbf{D}-1}$) or we come across a day j where $\mathbf{B}_j = 0$. This sum is the amount of bread we can eat if we swipe on day i! After calculating the amount of bread we could eat if we swiped on each day remaining in the semester, we can take the maximum of these values and output it as our answer.

When simulating the effect of using one meal card, we have to iterate over at most $\mathbf{D}$ days into the future. Since there are $\mathbf{N}$ possible days in the semester to swipe our meal card, this solution has a runtime of $O(\mathbf{ND})$.

Bonus Test Set A

While this test set still gives us a single meal card to use, it drastically increases the number of days in the semester, as well as the duration of the card's activation period. An $O(\mathbf{ND})$ solution won't cut it here; we'll need to be a lot faster!

MORE BREAD *Solutions available in Java, Python*

When we simulate the effect of swiping our meal card on a given day, we potentially iterate over the next $\mathbf{D}$ days to count how much bread we can eat with our swipe. This means that if we repeat this process for all $\mathbf{N}$ days, we could end up accessing each of the $\mathbf{B_i}$ values up to $\mathbf{D}$ times! This redundancy throttles the efficiency of our program—with some changes, however, it's possible to calculate all the same values as before while only accessing each $\mathbf{B_i}$ value a constant number of times!

We can begin with a simpler problem and assume $\mathbf{B_i} > 0$ for all i.

The following sums represent the amount of bread we can eat if we swipe our meal card on:

1. Day 1: $\mathbf{B_1} + \mathbf{B_2} + \mathbf{B_3} + \ldots + \mathbf{B_D}$
2. Day 2: $\mathbf{B_2} + \mathbf{B_3} + \ldots + \mathbf{B_D} + \mathbf{B_{D+1}}$
3. Day 3: $\mathbf{B_3} + \ldots + \mathbf{B_D} + \mathbf{B_{D+1}} + \mathbf{B_{D+2}}$

We can observe that if we already have b_i, the amount of bread we can eat by swiping on day i, then $b_{i+1} = b_i - \mathbf{B_i} + \mathbf{B_{i+D+1}}$. Importantly, this operation can be done in constant time! This is the core idea behind the sliding window technique: here, we "slide" a window of length $\mathbf{D}$ across the values $\mathbf{B_1}, \mathbf{B_2}, \ldots, \mathbf{B_N}$ one element at a time, using a single operation to update the sum of values within it every time it shifts. The window therefore represents an interval of $\mathbf{B_i}$ values claimed by swiping a meal card, with the left boundary of the window representing the actual day the card was used. With this technique, we can find the amount of bread eaten by swiping on each possible day in just $O(\mathbf{N})$ time!

If we reintroduce the possibility of $\mathbf{B_i} = 0$ for some values of i, we'll need to make some adjustments to our approach. Namely, we should clarify the definition of our window: while the left boundary represents the day we swipe our meal card, the right boundary represents the day we stop eating bread. This can happen either because $\mathbf{D}$ days have passed since our swipe, or because we've come across a zero $\mathbf{B_i}$ value.

With this in mind, we can adjust how our window slides across the $\mathbf{B_1}$, $\mathbf{B_2}$, …, $\mathbf{B_N}$ values. Specifically, when the right boundary of our window comes across a $\mathbf{B_i} = 0$, it should now be held in place during future iterations (while the left boundary continues to contract inwards). This is because any swipe that occurred before this day will stop eating bread upon reaching this zero $\mathbf{B_i}$ value. Eventually, the left and right boundaries will coincide (and both point to this zero $\mathbf{B_i}$ value). During the next iteration, move both the left and right boundaries to the right by one—future swipes now begin after this zero $\mathbf{B_i}$ value. Expand the right boundary out as far as you can, until the interval is $\mathbf{D}$ days wide or we come across another $\mathbf{B_j} = 0$. Afterwards, we can continue sliding the window as before!

We can visualize an example with $\mathbf{N} = 7$ days left in the semester, a single $\mathbf{K} = 1$ meal card that activates for $\mathbf{D} = 3$ days, and $\mathbf{B} = 5, 8, 6, 7, 0, 9, 3$ loaves of bread available at the cafeteria:

$\mathbf{B_1}$	$\mathbf{B_2}$	$\mathbf{B_3}$	$\mathbf{B_4}$	$\mathbf{B_5}$	$\mathbf{B_6}$	$\mathbf{B_7}$	Bread Eaten
5	**8**	**6**	7	0	9	3	**19**
5	**8**	**6**	**7**	0	9	3	19 - 5 + 7 = **21**
5	8	**6**	**7**	**0**	9	3	21 - 8 + 0 = **13**
5	8	6	**7**	**0**	9	3	13 - 6 = **7**
5	8	6	7	**0**	9	3	7 - 7 = **0**
5	8	6	7	0	**9**	**3**	**12**
5	8	6	7	0	9	**3**	12 - 9 = **3**

This approach is known as a variable-width sliding window, or as a two-pointer technique. Both the left and right boundary "pointers" access each $\mathbf{B_i}$ value a constant number of times, meaning this algorithm has an overall runtime of $O(\mathbf{N})$!

Bonus Test Set B

With us no longer being restricted to just one meal card, we'll need to write a much more flexible solution capable of determining how to make the most of our swipes!

MORE CARDS *Solutions available in Java, Python*

With multiple cards comes the decision of when to use each of them—although the expectation to make optimal decisions may be intimidating, we can save ourselves the logistical headache with the power of recursion! Namely, whenever a problem involves optimal decision-making, it's a good idea to see if we can use recursion to "cover all of our bases."

The general idea when it comes to recursion is breaking problems down into digestible pieces. What does this mean? Let's focus on a specific day i during the semester: on day i (assuming it's not within $\mathbf{D}$ days of a previous card swipe), we can either:

1. Use a meal card (assuming we have meal cards left), or
2. Not swipe a meal card, essentially doing nothing.

What actually happens in each scenario?

1. If we swipe a meal card, we accumulate all the bread for the next $\mathbf{D}$ days (or until day j, where $\mathbf{B}_j = 0$), and we now have one less card. Since we cannot swipe another card for the next $\mathbf{D}$ days, we can skip ahead to day $i + \mathbf{D}$.
2. If we do nothing, we keep the same number of cards and move into the next day, $i + 1$.

It's important to note that these two options **are the only things you can do on any given day**—any usage of our meal cards will either swipe a card on day i or not swipe a card on day i. If we can find the largest amount of bread we can eat throughout the semester—assuming we swiped a card right now, we could compare that value to the largest amount of bread we can eat throughout the semester—assuming we did nothing on our current day. The action that gives us more bread overall is the optimal one to take!

Now that we've broken up the problem into two smaller parts, we can write a function that helps us formalize everything mentioned so far. Let the function $most\_bread()$ output the largest amount of bread we can eat, given a relevant description of our "state." What defines a state can often be found in the parameters given to us in the problem statement! Namely, we start with $\mathbf{N}$ days left in the semester and $\mathbf{K}$ meal cards available to us. $\mathbf{D}$ and $\mathbf{B_1}$, $\mathbf{B_2}$, ..., $\mathbf{B_N}$ act more like constants, as they remain unchanged regardless of what we do.

Thus, if we have our function take in i, the current day (with 1 being the day the problem starts with and $\mathbf{N}$ being the last day of the semester), and k, the number of meal cards we currently have, the function could look something like this:

$most\_bread(i, k) = \max($
$\quad most\_bread(i + 1, k),$
$\quad most\_bread(i + \mathbf{D}, k - 1) +$ bread eaten by swiping on day i
$)$

This resembles what is normally called a recurrence relation, with the internal calls to $most\_bread()$ being subproblems. To find the amount of bread eaten by swiping on day i, you can directly simulate a swipe much in the same way as the main test set approach. To get the answer to the problem we were given, all we need to do is find $most\_bread(1, \mathbf{K})$!

If we call $most\_bread(1, \mathbf{K})$ as is, however, we'll just end up with a function infinitely calling itself as its parameter values spiral into oblivion. We need to establish some base cases first! Base cases are essentially inputs that the function memorizes the answer to—they act as the foundation for our tall stack of nested function calls, and tend to consist of states with trivial answers. A potential set of base cases for $most\_bread()$ could be:

- If $i > \mathbf{N}$, we've passed the last day in the semester. As such, $most\_bread()$ should appropriately output that we are able to eat zero loaves of bread.
- Similarly, if $k \leq 0$, we've run out of meal cards to use. $most\_bread()$ should also appropriately output that we are able to eat zero loaves of bread.

With our base cases in place, $most\_bread(1, \mathbf{K})$ can now give us our desired answer! The $\mathbf{N}$ possible values for i combined with $\mathbf{K} + 1$ possible values for k means that we have $O(\mathbf{NK})$ total possible states for $most\_bread()$. Evaluating a state takes $O(\mathbf{D})$ time—this comes from directly simulating a swipe on day i.

There's just one more detail to consider: as is, this recursive algorithm may repeat the same calculations multiple times. If $\mathbf{D} = 2$, a function call to $most\_bread(1, \mathbf{K})$ could call $most\_bread(4, \mathbf{K} - 1)$ twice:

1. Once by $most\_bread(1, \mathbf{K}) \rightarrow most\_bread(2, \mathbf{K}) \rightarrow most\_bread(4, \mathbf{K} - 1)$, and
2. Another by $most\_bread(1, \mathbf{K}) \rightarrow most\_bread(3, \mathbf{K} - 1) \rightarrow most\_bread(4, \mathbf{K} - 1)$.

In the worst case, despite only having $O(\mathbf{NK})$ distinct states, our algorithm ends up evaluating the outputs to $O(2^{\mathbf{N}})$ function calls!

The redundant calculations can be removed through either memoization or by using a dynamic programming implementation. With memoization, past inputs to $most\_bread()$ are "memorized"—their respective outputs are stored in a table or dictionary such that duplicate calls to $most\_bread()$ can have their answers directly looked up instead of recalculated!

With dynamic programming, outputs to subproblems are built from the "bottom-up," rather than "top-down" with recursion. Specifically, we can iterate backwards from day $\mathbf{N}$ to day 1, for increasing values of k from 0 to $\mathbf{K}$. By the time each state is evaluated, the values of the subproblems it relies on will have already been calculated; as a result, we can directly look up their values to evaluate our current state!

Through either of these techniques, each of the $O(\mathbf{NK})$ states is evaluated a constant number of times. Each state takes $O(\mathbf{D})$ time to evaluate, so the runtime of the algorithm is $O(\mathbf{NKD})$!

Bonus Test Set C

If you've been able to solve both bonus test sets A and B, now's your big chance! A solution that combines ideas from both subproblems will be able to be both fast and flexible enough to solve this final bonus test set.

oh my god we're UNSTOPPABLE *Solutions available in Java, Python*

The solution mentioned in bonus test set B is not as fast as it could be—since every subproblem calculates the value for $\text{bread eaten by swiping on day } i$ by directly simulating the effect of a swipe on day i, each subproblem takes $O(\mathbf{D})$ time to evaluate. If we could instead find a way to evaluate subproblems in constant time, we could drastically improve our algorithm's runtime from $O(\mathbf{NKD})$ to $O(\mathbf{NK})$!

We can do so by precomputing the values for $\text{bread eaten by swiping on day } i$ for each i ahead of time—this is acceptable because these values are independent from any of the parameters in our recurrence relation. Luckily, from bonus test set A, we already have an algorithm capable of efficiently accomplishing this! This solution is able to find the amount of bread eaten by swiping on each possible day in just $O(\mathbf{N})$ time—after which, our solution from bonus test set B will be able to look up values for $\text{bread eaten by swiping on day } i$ rather than calculate them on the spot. As a result, each subproblem can now be evaluated in constant time, meaning the runtime for running our solution from bonus test set B is reduced to $O(\mathbf{NK})$!

This means that, precomputing included, our unified algorithm has an overall runtime of $O(\mathbf{N}) + O(\mathbf{NK}) = O(\mathbf{NK})$, and is both fast and flexible enough to pass all test cases!